Direct and inverse translators between FLogic and Ontolingua in the context of ODE and the (KA)^2 initiative: a case of study

Oscar Corcho, Asunción Gómez-Pérez
Facultad de Informática
Universidad Politécnica de Madrid
Campus de Montegancedo s/n
Boadilla del Monte, 28660. Madrid. Spain.
Tel: (34-1) 336-74-39, Fax: (34-1) 336-74-12
Email: {ocorcho, asun}@delicias.dia.fi.upm.es

Abstract

This paper focuses on our experiences on translating ontologies from one ontology implementation language, FLogic, into another one, Ontolingua, in the framework of Methontology and ODE. Rather than building "ad hoc" translators between languages and to carry out the translation at the implementation level, our option consists of translating from ODE intermediate representations, which are at the knowledge level. So, we have built direct translators from ODE intermediate representations to Ontolingua and FLogic, and we have also built reverse translators from these two languages to ODE intermediate representations. Expressiveness of the target languages is the main feature to analyze when generating automatically ontologies from ODE intermediate representations. In this paper, we just analyze the expressiveness of Ontolingua and FLogic for creating classes, instances, relations, functions and axioms, which are essential components in ontologies. The motivation for this analysis can be found in the (KA)^2 initiative and can be easily extended to any other domains and languages.

1. Introduction

At present there exist lots of ontologies coded in many different languages (Ontolingua (Gruber,93), KIF[1](Genesereth et al, 92), CycL[2](Lenat et al, 90), FLogic(Kifer et al, 95), LOOM (MacGregor,91)...). All this ontologies have in common a feature: they all have just been developed and coded using these languages (using directly the target language syntax or using editors provided by the developers of these implementation languages); so, it has been made a lot of work at the implementation level, instead of the knowledge level. Working at the implementation level causes several problems (Fernández et al., 99). One of them is the fact that ontology developers (who are unfamiliar with or simply inexperienced in the languages in which ontologies are coded) may find difficult to understand implemented ontologies or even to build a new ontology, because traditional ontology tools focus too much on implementation issues rather than on questions of design. The other is that ontology conceptual models are implicit in the implementation codes and that the expressiveness of the implementation language condition the conceptual model already created.

---

[1] http://logic.stanford.edu/kif/dpans.html Knowledge Interchange Format

[2] http://www.cyc.com/cycl.html Features of the CycL Language

To solve both problems, the Ontology Design Environment (ODE) (Blázquez et al., 98) is being built. Currently, the main advantage of ODE conceptualization module is that the ontologist develops the ontology at the knowledge level using a set of intermediate representations (IRs) that are independent of the target language in which the ontology will be implemented. Once the conceptualization is complete, the code is generated automatically using ODE translators. ODE's big advantage is that it enables specification of ontologies at the knowledge level, delegating implementation to fully automated code generators. Current translators include: Ontolingua, FLogic and a relational database. So, non-experts in the languages in which ontologies are implemented could specify and technically evaluate ontologies using this environment.

As set out before, many ontologies exist nowadays, and they are coded in many different languages. It would be great to have every existing ontology in any desired language without having to code it again. Translators are not enough to carry out this task, due to the following problems:

- Incorrect logical structure of the concepts represented in the ontology, because of a bad approach to its development. That makes it difficult to reuse knowledge from the ontology.

- Different *codification styles* for ontology's components. This problem specially arises in Ontolingua ontologies, because ontologies can be defined in KIF; using the Frame Ontology vocabulary (Gruber, 93), or using the Ontolingua Server (Farqhuar et al, 96) editor.

- Similar definitions are defined using different patterns. To minimize the semantic distance between sibling concepts (Arpírez et al., 98), similar concepts should be defined using the same primitives to give a clearer understanding of the ontology.

- The names of the terms are not standardized.

- Some pieces of information can be easily forgotten.

- It's difficult to evaluate the consistence and completeness of information represented in the ontology.

Ontological reengineering (Gómez-Pérez et al,99) is proposed as a process which starts taking information from one or more existing coded ontologies, restructure these pieces of information guided by a suitable methodology and criteria, and therefore generate code according to some established and broad-accepted guidelines. Reengineering can be thought of a good process for increasing the readability of existing ontologies, by changing old code with other definitions which follow the same patterns (this use of reengineering can be clearly taken into account in the case of definitions in Ontolingua).

2. Motivation

This paper is placed in the framework of the $(KA)^2$ initiative[3] (Benjamins et al, 98), whose goal is to model the knowledge-acquisition community. To model the

[3] Home page of the $(KA)^2$ initiative is at http://www.aifb.uni-karlsruhe.de/WBS/broker/KA2.html

community we use ontologies. The current conceptual model of the $(KA)^2$ community consists of seven related ontologies: an organization ontology, a project ontology, a person ontology, a publication ontology, an event ontology, a research-topic ontology and a research-product ontology. The majority of these ontologies have been built by the coordinating agents. However, the development of the research topic ontology has being carried out by groups of researchers situated at different locations. Each group submitted its part of the research topic ontology to the coordinating agents that integrated all of them. $(KA)^2$ ontologies are built jointly by geographically distributed groups in FLogic.

The development of the $(KA)^2$ ontology is divided in two phases. The first part is concerned with the development of its conceptual structure, and the identification of its main concepts, taxonomies, relations, functions, attributes and axioms. The second part corresponds with the addition of knowledge about specific instances.

A first release of this ontology was built in FLogic, which is the ontology that Ontobroker (Fensel et al., 98) uses. A decision was made to translate the whole ontology to Ontolingua to make it accessible to the entire community through the Ontolingua Server. Since concepts as far as authors, research centers, publications, etc. had been represented in a single ontology, the option of directly translating from FLogic to Ontolingua was ruled out and it was decided to carry out an ontological reengineering process followed the guidelines presented in (Blázquez et al., 98). First, we obtained the $(KA)^2$ conceptual model attached to the FLogic ontology manually by reverse engineering. Second, we restructured it (according to the modularity criterion) using the Ontology Design Environment (ODE) conceptualization modules and we got a new conceptual model. Finally, we converted the restructured $(KA)^2$ conceptual model to Ontolingua using ODE forward translators. Figure 1 shows the $(KA)^2$ ontology life cycle. Note that, in this case, our source of knowledge was an ontology that had already been implemented in the FLogic language. The current version of the Ontolingua ontology can be found at the European mirror site in Madrid of the Ontolingua Server of Stanford University[4]. Login as ``ontologias-ka2" with password ``adieu007".

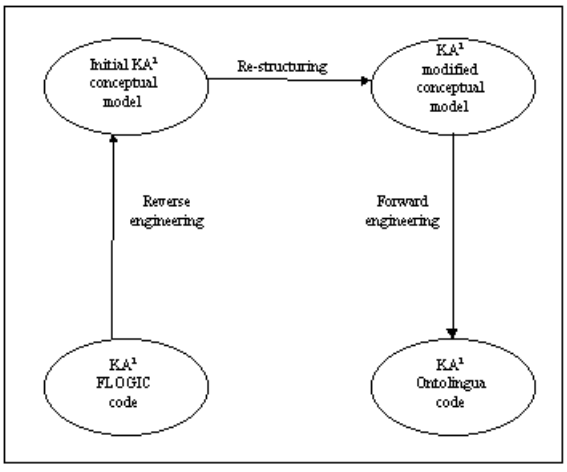

Figure 1. $(KA)^2$ ontology life cycle (Blázquez et al., 98)

Usually, ontologies are developed by individuals or groups of individuals located at the same place, working in the same project. This is not the scenario of the $(KA)^2$

[4] http://www-ksl-svc-lia.dia.fi.upm.es:5915

ontologies, which are being developed in a joint effort by a group of people at different locations. Apart from the problem of group dynamics and processes in ontology construction, ontologies built in a jointly and distributively way have the problem of updates when some parts of the ontology are implemented in different languages at different times. In this case, it becomes really important to have translators in order to automatically generate code from the source languages to the target languages and viceversa. Differences in expressiveness of implementation languages is the first problem to solve. A translator must generate the same information expressed in the source code when translating to the target language. When translations have to be made among n languages, 2^n different translators are required. That is the reason why interchange formats like KIF (Genesereth et al., 92) and PIF (Lee et al., 96) were proposed to interchange knowledge and processes respectively between heterogeneous and independent applications.

Given the already known advantages of building ontologies at the knowledge level and the advantages of using the intermediate representations proposed at (Fernández et al., 99), our approach is to use such intermediate representations as an interchange format. Consequently, direct translators from ODE intermediate representations to target languages used to implement ontologies are required, as well as reverse translators from such languages to ODE intermediate representations. In fact, definitions from the implementation level were taken into the knowledge level by a Reverse Engineering process (Gómez-Pérez et al., 99). Inverse translators will fill in the intermediate representations of ODE from the initial source code. Once we have this knowledge in ODE, we can transform the conceptual model of the ontology using ODE conceptualization module. After this, by a forward engineering process, direct translators generate the chosen target codes.

In order to develop direct and reverse translators we must analyze the differences (not only in syntax, but specially in semantics) of different ontology implementation languages. This paper focuses on the differences between Ontolingua and FLogic, presenting the main difficulties we must face when trying to express the same knowledge in both languages.

3. Short description of implementation languages

3.1. Ontolingua

Ontolingua is a language based on KIF (Genesereth et al., 92) and on the Frame Ontology (Gruber, 93), and it is the ontology-building language used by the Ontolingua Server (Farqhuar et al., 98).

KIF (Knowledge Interchange Format) is developed to solve the problem of heterogeneity of languages for knowledge representation. It provides for the definition of objects, functions and relations. KIF has declarative semantics and it is based on the first-order predicate calculus, with a prefix notation. It also provides for the representation of meta-knowledge and allows for the representation of non-monotonic reasoning rules. KIF sentences are composed of lists which have a relation constant as their first term and an arbitrary number of terms following it, and logical operations among several sentences.

As KIF is an interchange format, it is very tedious to use for specification of ontologies per se, but there is the frame ontology (Gruber, 93) built on top of it which allow an ontology to be specified following the paradigm of frames. The Frame Ontology is a knowledge representation ontology for modeling ontologies under a frame-based approach. It was built on the basis of KIF and a series of extensions to this language. Terms like: class, instance, subclass-of, instance-of, etc are included in this ontology.

Since the Frame Ontology is less expressive than KIF, that is, not all of the knowledge that can be expressed in KIF can be expressed using the *Frame-Ontology*, Ontolingua allows to include KIF expressions inside of definitions based on the *Frame-Ontology*. So, the Ontolingua language allows to build ontologies in any of the following three manners: (1) using exclusively the Frame Ontology vocabulary (it is not possible to represent axioms); (2) using KIF expressions; (3) using both languages simultaneously,

```
(Define-Class Elements (?Elements)
"It is a substance that are made up only by atoms with the same number of protons."
:def
  (and
    (Superclass-Of ?Elements  Non-reactive-elements  Reactive-elements)
    (Has-One ?Elements Atomic-Number)
    (Has-Some ?Elements Chemical-Group))
:axiom-def
  (Exhaustive-Subclass-Partition Elements
  (Setof Non-reactive-elements Reactive-elements)))
```

a) Using the Frame-Ontology

```
(Define-Frame Channel
        :Own-Slots
        ((Instance-Of Class)
         (Subclass-Of Electronic-Device)
         (Documentation
            "The notion of a channel as a device is counterintuitive, but it makes it easier to talk
            about the characteristics of a device when the device is using that channel."))
        :Template-Slots
        ())
```

b) Using the Ontolingua Server browser

```
(defrelation Individual
        (?x)
        :=
        (and (Thing ?x) (not (Class ?x)) (not (Set ?x))))
(Subclass-Of Individual Thing)
(Class Individual)
(Arity Individual 1)
(Documentation Individual
 "An individual is something that isn't a set, but that can be a member of a set. ")
```

c) Using KIF

```
(define-class SIMPLE-SET (?x)
 "A simple set is a set that can be a member of another set."
 :iff-def (and (set ?x)
           (bounded ?x))
 :constraints (thing@okbc-ontology ?x)
 :issues ("The constraint that simple-set is a subclass-of thing is specified as part of the
     definition of THING, but it needs to be here, too, for bootstrapping reasons."
       (:source "KIF Version 3.0 Specification")))
```

d) Using Frame Ontology + KIF

Figure 2. Different ways of implementing classes in Ontolingua

depending on ontology developer preferences. In any case, the Ontolingua definition is composed of a heading, an informal definition in natural language and a formal definition written in KIF or using the frame ontology vocabulary. Eventually it has been developed the *OKBC-Ontology*, which improves some aspects of the *Frame- Ontology*.

Ontolingua ontologies are kept at the Ontolingua Server, which allows users to build ontologies using the Ontolingua editor or to import ontologies already built using a text editor. *Figure 2* presents different ways of implementing classes in Ontolingua, taken from existing ontologies in the server.

3.2. FLogic

FLogic (Kifer et al., 95) is an acronym for *Frame Logic*. FLogic is a language which integrates frame-based languages and first-order predicate calculus. It accounts in a clean and declarative fashion for most of the structural aspects of object-oriented and frame-based languages. These features include object identity, complex objects, inheritance, polymorphic types, query methods, encapsulation, and others. In a sense, F-logic stands in the same relationship to the object-oriented paradigm as classical predicate calculus stands to relational programming. F-logic has a model-theoretic semantics and a sound and complete resolution-based proof theory. A small number of fundamental concepts that come from object-oriented programming have direct representation in F-logic; other, secondary aspects of this paradigm are easily modeled as well.

4. Main components of an ontology.

Ontologies provide a common vocabulary of an area and define –with different levels of formality- the meaning of the terms and the relations between them. Knowledge in ontologies is formalized using five kind of components: classes, relations, functions, axioms and instances (Gruber, 93). Classes in the ontology are usually organized in taxonomies. Sometimes, the definition of ontologies have been diluted, in the sense that taxonomies are considered to be full ontologies (Studer et al., 98).

- Concepts are used in a broad sense. They can be abstract or concrete, elementary (electron) or composite (atom), real or fictitious. In short, a concept can be anything about which something is said and, therefore, could also be the description of a task, function, action, strategy, reasoning process, etc.
- Relations represent a type of interaction between concepts of the domain. They are formally defined as any subset of a product of n sets, that is: R: C1 x C2 x ...x Cn. Examples of binary relations are: subclass-of and connected-to.
- Functions are a special case of relations in which the n-th element of the relationship is unique for the n-1 preceding elements. Formally, functions are defined as: F: C1 x C2 x ... x Cn-1 → Cn. Examples of binary functions are Mother-of and square, and an example of a ternary function is price-of-a-used-car that calculates the price of a second-hand car depending on the car-model, manufacturing date and number of kilometers.
- Axioms are used to model sentences that are always true.
- Instances are used to represent elements in the domain.

This paper has focused on a detailed analysis of the expressiveness of how Ontolingua and FLogic allow to specify such components.

```
(Define-Class class-name(?class-name)
  "Description"
(:Def | :Iff-Def)
(and
   {(father-class-name ?class-name)}
   ((Superclass-Of ?class-name
        {child-class-name})]
   ((Type-Of ?class-name {instance-name})]
   {(Has-At-Least attribute-name ?class-name n)}
   {(Cardinality attribute-name ?class-name
n)})
(:Axiom-Def
    ((Exhaustive-Decomposition ?class-name
        (Setof {child-class-name}))]
    (Disjoint-Decomposition ?class-name
        (Setof {child-class-name}))]])
```

a) Ontolingua

```
{class-name :: father-class-name.}
class-name (
({attribute-name attribute-name };}]
({attribute-name -> value ;}]
({attribute-name =>> type ;}]
({attribute-name => type;}]
].

{child-class-name :: class-name.}

{instance-name : class-name.}
```

b) FLogic

Figure 3. Patterns of a class

```
(Define-Class Research-Topic( ?Res-Topic)
  "No description"
:Def
(and
   (Superclass-Of ?Res-Topic
        KA-Through-Machine-Learning
        Reuse
        Specification-Languages
        Validation&Verification
        Knowledge-Management
        KA-Methodologies
        Evaluation-of-KA
        Knowledge-Elicitation)
  (Has-At-Least Approaches ?Res-Topic 1)
  (Cardinality Date-of-last-modification ?Res-
Topic 1)
  (Has-At-Least Related-Topics ?Res-Topic 1)))
```

a) Ontolingua

```
ResearchTopic : :Object.
ResearchTopic(
        description -> "No description" ;
        Approaches =>> Topic ;
        DateOfLastModification => DATE ;
        RelatedTopics =>> ResearchTopic].

KAThroughMachineLearning : :ResearchTopic.
Reuse : :ResearchTopic.
Specification-Languages : :ResearchTopic.
Validation&Verification : :ResearchTopic.
Knowledge-Management : :ResearchTopic.
KA-Methodologies : :ResearchTopic.
Evaluation-of-KA : :ResearchTopic.
Knowledge-Elicitation) : :ResearchTopic.
```

b) FLogic

Figure 4. Examples of class

4.1. Classes and Taxonomies

Formally, an ontology consists of a set classes that are organized in taxonomies. A class stands for a type of objects in our universe. Defining a class in both Ontolingua and FLogic is not difficult and they can follow the patterns presented in *figure 3*.

As we said before, Ontolingua allows to code classes with several patterns. We have chosen *Define-Class*, pattern definition, as it is the most used pattern to create classes. As we can see, the following information can be represented inside a class: documentation, its superclass(es), its subclass(es), its instance(s), and its attributes,

```
(Define-Relation relation-name ({?class-name})
"Description"
:def
   (and
         {(class-name ?class-name)}
         {(value-type ?value-type)}
         ((>=?value-type min-value)
          (=<?value-type max-value)])
(:Axiom-Def
         ((Minimum-Slot-Cardinality relation-name min-cardinality)]
         ((Maximum-Slot-Cardinality relation-name min-cardinality)]
         ((Inverse relation-name inverse-relation-name)]])
```

Ontolingua

Figure 5. Pattern of a relation definition in Ontolingua

```
(Define-Relation Developed-by (?Product ?Organization)
"Product which is developed by an organization"
:def
   (and
         (Product ?Product)
         (Organization ?Organization)))
```

a) Ontolingua

```
Product(DevelopedBy => Organization].
```

b) FLogic

Figure 6. Examples of an attribute definition

along with their minimum cardinality or a fixed one. It can also be described subclass-partitions and/or exhaustive subclass partition[5].

Ontolingua description (*figure 3.a*) cannot be coded just in one definition in FLogic. In FLogic (*figure 3.b*), class hierarchy is described with*::*, while the *instance of* relation is described with *:*. Attributes are also described with their name and type. Their cardinality is determined by the kind of arrow: double arrow means that it is a multivalued attribute, and a single arrow means that it is a single-valued attribute. The difference between double and single arrows lays on the kinds of attributes that we are describing: the former ones are used for instance attributes, the other ones are used for class attributes, with their value.

Figure 4 shows an example, taken from one of the ontologies created for the $(KA)^2$ initiative. We pretend to show differences in syntax and content of the same definition in both implementation languages. We describe the class of *research topics*, which has the following subclasses: *KA Through Machine Learning, Reuse, Specification Languages, Validation&Verification, Knowledge Management, KA Methodologies, Evaluation of KA* and *Knowledge Elicitation*. We also define an attribute, called *Approaches*, which can have multiple values, another attribute, *Date of last modification*, which can only have a single value, and the last one, *Related Topics*, which can have multiple values as well.

Differences in syntax are not as important as differences in semantics. As set out below, there are some differences in expressiveness between them:

[5] A subclass partition of C is a set of subclasses of C that are mutually disjoint. An exhaustive subclass of C is a set of mutually disjoint classes that covers C. Every instance of C is an instance of exactly one of the subclasses in the partition.

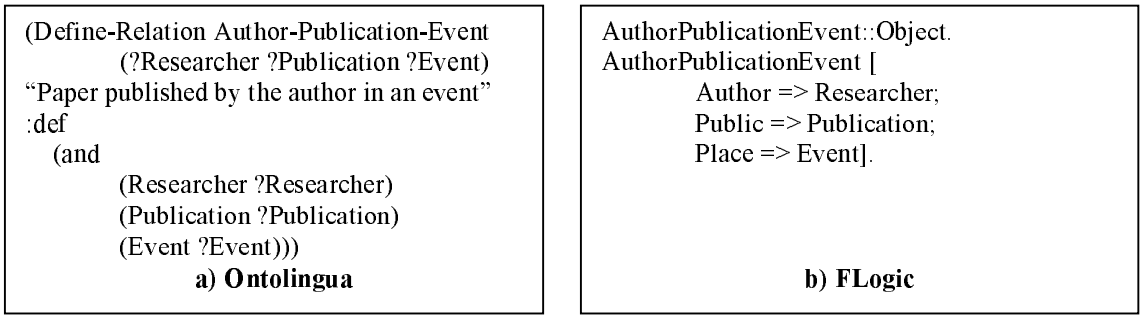

Figure 7. Examples of a relation definition

- Ontolingua allows to add new facets (apart from the existing ones) to slots attached to a class. FLogic just allows to set maximum cardinality and value type facets.

- Slot cardinality can be fixed completely in Ontolingua (we can set minimum and maximum cardinality with any natural number, e.g., between 2 and 5). FLogic makes a distinction in cardinality depending on the kind of arrow used to describe an attribute, in such a way that we can only set the maximum cardinality with value **1** or value **several**. Minimum cardinality is fixed (value **0**) and cannot be modified.

- Methods can be defined in FLogic, as it is commonly used in the object-oriented paradigm, or with daemons in frames.

- Subclass partitions and exhaustive subclass partitions can be defined in Ontolingua. These definitions are not allowed in FLogic.

- In Ontolingua, documentation slot is defined as a default. In case we want this slot in a FLogic class, it must be clearly set.

- Class attributes can be defined in Ontolingua by using the macro *Define-Frame*. FLogic has just one way to describe a class, and class attributes can be described using a single arrow.

Attributes are coded in a different way in Ontolingua and FLogic. While FLogic includes its definition in the same structure where the class is described, in Ontolingua the definition of attributes must be made in a different structure. In *figure 5* we present how to do it with *Define-Relation*.

In figure 6, we present an example from one of the $(KA)^2$ ontologies, where it is set out that the a *product* is *developed by* an *organization*.

4.2. Relations

Relations represent types of interaction between concepts of the domain. Ontolingua allows to define n-arity relations, involving different concepts of the ontology. These kind of relations does not exist in FLogic, as this mechanism is not directly supported; and we must use first-order calculus instead. Therefore, if we want to define them in FLogic, relations should be represented as classes, where the attributes of the class are the elements involved in it, as it is shown in *figure 7*.

(**Define-Function** function-name ({**?**variable-name}) **:->** **?**output
"Description"
:def (and
{(**Instance-Of** ?variable-name class-name)}
{(atrribute-name ?variable-name ?attribute-name)}
(logic-expression))
(**:lambda-body** (aritmethic-expresion)])

a) Ontolingua

class-name(
method-name@{class-name}=>>class-name].

FLogic expression.

b) FLogic

Figure 8. Patterns of a function definition

(Define-Function Joint-Works (?x ?y) :-> ?z
"Defines the paper written by x and y"
:def (and
(Instance-Of ?x Researcher)
(Instance-Of ?y Researcher)
(Instance-Of ?z Paper)
(Papers ?x ?z) (Papers ?y ?z)))

a) Ontolingua

Researcher::Person(
jointWorks@Researcher=>>Paper].
X(jointWorks@Y->>Z] ← X:Researcher ∧
Y:Researcher ∧
X(Papers->>Z] ∧
Y(Papers->>Z].

b) FLogic

Figure 9. Examples of a function definition

One advantage of Ontolingua is that it is possible to define the name of the inverse relation without having to write an axiom, as FLogic needs.

Ontolingua's syntax for relations was pointed out in *figure* 5. *Figure 7* shows a relation where a *researcher* publishes a *publication* in an *event*.

4.3. Functions.

Functions are a special case of relations in which the n-th element of the relationship is unique for the n-1 preceding elements. FLogic only allows to define functions within the body of a class.

In order to define a function in Ontolingua, we must give it a name, describe its arguments, its output and a description. The body of the function must state the classes to which its arguments belong, attributes involved from this classes, and optionally a lambda body, where we can describe (using KIF syntax), the operations needed to calculate the output value. See *figure 8*.

Example in *figure 9* shows a function definition for class Researcher, which returns a paper that its arguments (two researchers) have written in common:

Next comments can be made:

- FLogic only allows to define functions related to a class, and not as independent units. So, functions are always connected with at least one attribute, which seems to be the main attribute.

- FLogic definition for a function takes its first argument from the class where it is defined. In Ontolingua, this first argument must be specified. The other involved arguments must be specified in both definitions.

(**Define-Axiom** axiom-name "Description" := (KIF-expression)) **a) Ontolingua**	*First-order calculus expression using FLogic syntax* **b) FLogic**

Figure 10. Patterns of an axiom definition

```
(Define-Axiom Carried-Out-By
 "It defines the relationship between organizations and projects"
:= (forall ?x
      (forall ?y
       ( <=>
          (and (Instance-Of ?x Organization) (Instance-Of ?y Publication) (Carries-Out ?x ?y))
          (Carried-Out-By ?y ?x)))))
```

a) Ontolingua

```
FORALL Organiz1,Project1
Organiz1:Organization(carriesOut ->> Project1] <-> Project1:Project(carriedOutBy ->> Organiz1].
```

b) FLogic

Figure 11. Examples of an axiom definition

4.4. Axioms

Axioms are used to model sentences that are always true. They can be used as well to constraint property and role values for classes or instances. Ontolingua and FLogic are primarily based on first-order calculus. Both languages allow to use axioms not only for defining constraints on attribute values, but also for stating other facts.

As they are based on first-order order calculus, the same expressiveness can be achieved with both of them. Differences can just be found in the readability of expressions.

Figure 10 shows the syntax of axioms in both languages. Example in *figure 11* defines an axiom (called *Carried-Out-By* in Ontolingua's code, and without a name in FLogic), which states that *Carried Out By* is the inverse relation of *Carries Out*, and takes as arguments a *publication* and an *organization*.

- Ontolingua allows to give names and descriptions to axioms (this feature is useful in order to reference them), while this is not allowed in FLogic.

- We can achieve the same expressiveness in axioms, as both of them are based of first-order calculus.

```
(Define-Individual instance-name(class-name)
"Description"
:axiom-def
 (and
  {(instance-attribute-name instance-name value)}))
```

a) Ontolingua

```
instance-name : class-name(
        ({attribute-name ->> {value};}]
        ({attribute-name -> value ;}]
].
```

b) FLogic

Figure 12. Patterns of an instance definition

```
(Define-Individual KEML(Workshop)
"No description"
:axiom-def
  (and
        (Workshop-Series KEML "KEML")
        (Event-Title KEML "Knowledge-
Engineering-Methods&Languages")))
```

a) Ontolingua

```
KEML:Workshop(
        WorkshopSeries → "KEML";
        EventTitle →"Knowledge Engineering
Methods&Languages"].
```

b) FLogic

Figure 13. Examples of an instance definition

4.5. Instances

Every ontology implementation language allows to easily define instances, which are also called individuals in some languages.

Ontolingua supports several ways of defining instances. We have chosen the macro *Define-Individual*, where we will set values for the instance attributes declared in the class definition. FLogic describes instances using the same style used for classes

Instance descriptions in Ontolingua and FLogic are very similar (See *figure 12*): we must specify the class to which it belongs and instance attributes values.

Figure 13 presents an instance, called *KEML*, which belongs to class *Workshop*, and whose attribute values for the instance attributes *Workshop-Series* and *Event-Title* are *KEML* and *Knowledge Engineering Methods & Languages*. Although FLogic syntax is clearly more legible than Ontolingua's one, this will not generate any problems in automatic translation from ODE, as the semantics is the same:

- Values of instance attributes must be defined in Ontolingua by using a KIF relation, which consists of an instance attribute name (relation name), an instance name and a value. FLogic define them by its name, a single or double headed arrow, and its value(s).

- To set *n* values for the same attribute, in Ontolingua we must write *n* times the structure described above, while FLogic allows to set them in one expression. Let´s take an attribute *Colour* from a car which has two colours: blue and white. We would say: *(Colour My-Car Blue) (Colour My Car White)* in Ontolingua, while FLogic expression would be: *Colour->>{Blue,White}*.

- FLogic does not include by default the instance attribute *Description*. We have just defined a class attribute called *Description*, so we could use for instances an instance attribute called *InstanceDescription* which could be defined in class *Object* (the superclass of all classes), as we cannot use attribute *Description* again.

4.6. Inference

FLogic's inference engine allows to obtain directly attributes inherited throughout a concept taxonomy. However, Ontolingua does not have an inference engine. To reason with Ontolingua ontologies, a GFP module must be built (which is not an easy task) in order to get inherited attributes.

When translating between two languages which support different types of inheritance (monotonic and non-monotonic) we may find difficulties, derived from differences in semantics, when the value of an inherited attribute is defined several times throughout the class hierarchy. Both Ontolingua and FLogic support non-monotonic inheritance. Therefore, this problem will not arise.

4.7. Expressiveness of Ontolingua and FLogic

Examples of code in both languages have shown the main advantages and disadvantages of each one. Next, we sum up the main similarities and differences in expressiveness found in that analysis.

Both languages allow:

- Creating hierarchies of concepts.
- Defining classes, and the definition of instance attributes attached to the class, as well as the type of values for each attribute.
- Defining instances.
- Stating facts by defining axioms which use first-order calculus.

Ontolingua's main advantages are:

- When coding an ontology in Ontolingua, we can make use of primitives provided by the frame-ontology (a knowledge representation ontology) to create all the classes, relations, functions, instances and axioms of the ontology. In contrast, FLogic lacks from a knowledge representation ontology.
- Every element in the ontology can be named and formally and informally described. That is, the name and description slots are always present in Ontolingua.
- Ontolingua provides more facets than FLogic. See them at the frame ontology.
- Subclass partitions and exhaustive subclass partitions of a given class can also be directly defined inside the definition of the class. To state this feature in FLogic, an axiom must be written.
- n-arity relations can be defined in a frame-based style.

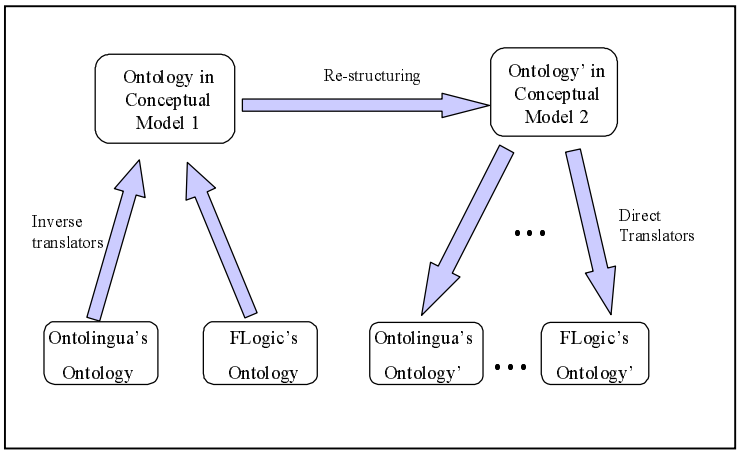

Figure 14. Direct and inverse translators in ODE

FLogic's main advantages are the following:

- In FLogic, there's just a unique way to define elements in the ontology. Ontolingua's elements can be coded using several different structures (with the *Frame Ontology*, using KIF, merging *Frame Ontology* and KIF, and using the Ontolingua Server Editor).

- FLogic code can be executed and FLogic inference engine provides inheritance. To execute an Ontolingua ontology, a GFP module is required.

5. Conclusion

This analysis helped us on the task of identifying patterns in two languages for implementing ontologies: Ontolingua and FLogic. This is necessary for ODE direct and inverse translators modules, as we set out below.

ODE's big advantage is that it enables *specification of ontologies at the knowledge level, delegating implementation to fully automated code generators*. So, ontologies are built filling in tables and graphs which are the inputs to the direct translators. The translators use a series of transformation rules, which contain the kind of structures (patterns) pointed out before, to generate the ontology in a given target code. For a detailed description, see (Fernández et al., 99). Using the translator that converts the conceptualization into an implementation, error-free code is generated, which dramatically cuts the time and effort involved in the implementation of ontologies

Inverse translators follow the same structure that direct translators, that is, inverse translators take information from existing ontologies (coded in any language) and fill in ODE's intermediate representations by using a series of inverse transformation rules. The complete process can be seen in *figure 14*.

When building inverse translators, the main advantage of FLogic (in contrast with Ontolingua) is that information can only be expressed in a unique way; therefore all of the knowledge coded in an ontology can be taken into ODE's intermediate representations. Ontolingua allows to define the same contents in several ways, and that makes it difficult to build an inverse translator which covers all the possible definitions.

Currently, we are performing a statistical study of the syntax used in ontologies in the Ontolingua Server.

At present ODE translators module includes:

- Direct translators from the set of IRs to FLogic and Ontolingua.
- Inverse translators from FLogic and Ontolingua to the set of IRs. Ontolingua's inverse translator is incomplete, as it just covers the most frequently used patterns.

6. Acknowledgements

This research is supported by the program "Acciones Integradas Hispano-Alemanas", reference HA1998-0002, funded by the "Ministerio de Educación y Cultura" in Spain

7. References

Arpírez, J.;. Gómez-Pérez, A.; Lozano, A.; Pinto, S. *(ONTO)^{2}Agent: An ontology-based WWW broker to select ontologies.* . Workshop on Applications of Ontologies and PSMs. Brighton. England. August 1998. pp. 16-24.

R. Benjamins, D. Fensel. Community is knowledge in (KA)^2. In B. R. Gaines and M. A. Musen, editors. Proceedings of the 11th Banff Workshop on Knowledge Acquisition, Modelling and Management (KAW98). Pages KM-2-1-KM-2-18. Alberta, Canada, 1998.

Blázquez M., Fernández M., García-Pinar J. M., Gómez-Pérez A., *Building Ontologies at the Knowledge Level using the Ontology Design Environment*, Proceedings of the Eleventh Knowledge Acquisition Workshop, KAW98, Banff, 1998.

Chaudhri Vinay K., Farquhar Adam, Fikes Richard, Karp Peter D., Rice James P. *The Generic Frame Protocol 2.0* , Technical Report, Stanford.1997.

Farquhar A., Fikes R., Rice J., *The Ontolingua Server: A Tool for Collaborative Ontology Construction*, Proceedings of the 10th Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Alberta, Canada, PP. 44.1-44.19, 1996.

Fensel, D, Decker, S. Erdman M. Studer, R. *Ontobroker: The Very High Idea.* In Proceedings of the 11th International Flairs Conference (FLAIRS-98), Sanibal Island, Florida, May 1998.

Fernández, M.; Gómez-Pérez, A.; Pazos, J.; Pazos, A. Building a Chemical Ontology using methontology and the ontology desing environment. IEEE Intelligent Systems and their applications. !4 (1):37-45. 1999.

Genesereth M., Fikes R., Knowledge Interchange Format, Technical Report, Computer Science Department, Stanford University, Logic-92-1, 1992.

Gómez-Pérez, A.; Rojas. M.D. *Ontological Reengineering for reuse.* EKAW99.

Gruber, T. A translation Approach to portable ontology specification. Knowledge

Acquisition. 5: 199-220. 1993.

Kifer M., Lausen G., Wu J., *Logical Foundations of Object-Oriented and Frame-Based Languages*, Journal of the ACM, 1995.

Lee, J.; Grüninger, M.; Jin, Y.; Malone, T.; Tate, A.; Yost, G.*; Process Interchange Format (PIF)*. Workshop on Ontological Engineering. ECAI'96. Budapest. Hungary: 65-76, 1996.

Lenat D.B., Guha, R V. Building Large Knowledge-based systems. Representation and Inference in the Cyc project. Addison-Wesley, Reading, Massachusetts. 1990.

MacGregor, R. Inside the LOOM classifier. SIGART bulletin, 2(3):70-76. June, 1991.

R. Studer, V.R. Benjamins, D. Fensel. Knowledge Engineering: Principles and Methods. Data & Knowledge Engineering. 25: 161-197. 1998.